Dynamical Systems Final Project

CID: 02044064

March 21, 2025

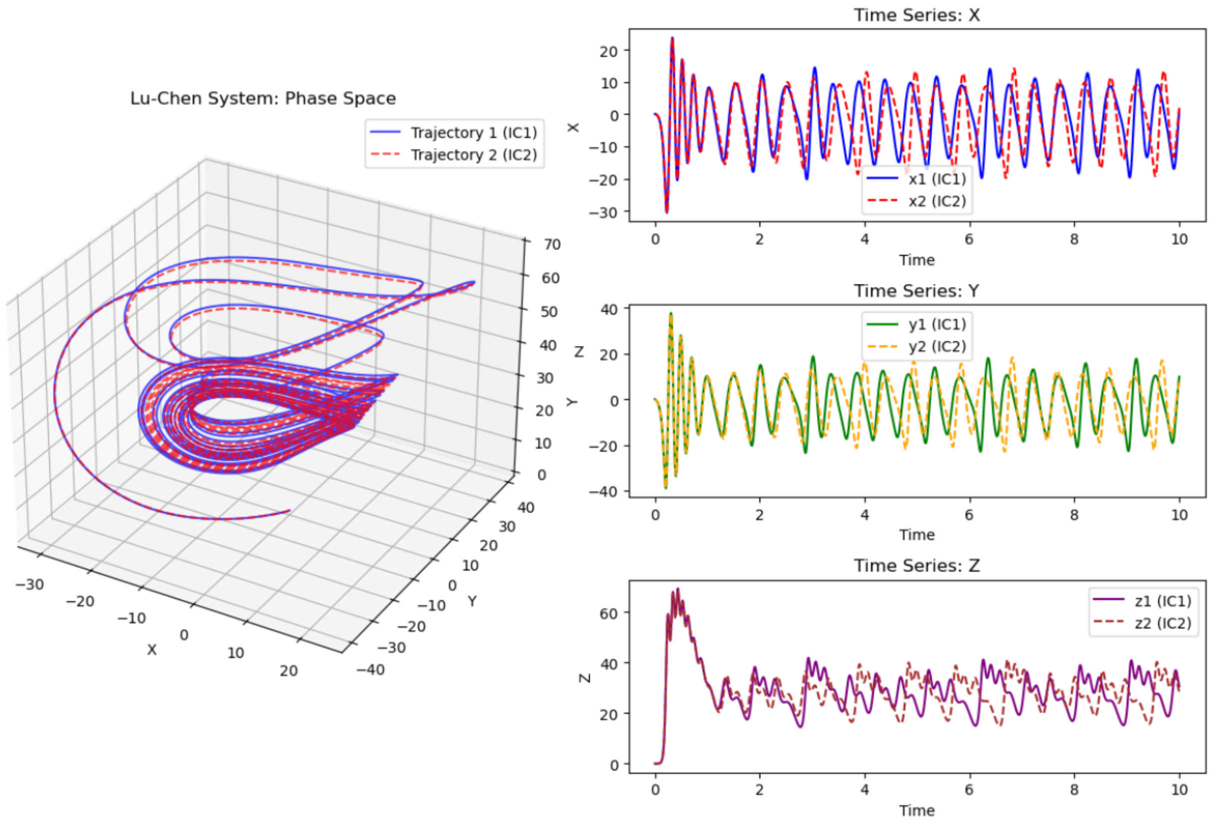
Part 1: Bifurcation and Chaos

1. Phase Space and Sensitivity to Initial Conditions

The code below is based on lecture 7: Chaos and Strange Attractors.

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp

# Define the Lu-Chen system
def lu_chen(t, state, a, b, c, u):
    x, y, z = state
    dxdt = -a * x + a * y
    dydt = -x * z + c * y + x + u
    dzdt = x * y - b * z
    return [dxdt, dydt, dzdt]

# Parameters
a, b, c, u = 25.90, 2.98, 21.30, -15.28

t_span = (0, 10)
t_eval = np.arange(t_span[0], t_span[1], 0.001)  # Time points

# Initial conditions
x0, y0, z0 = 0, 0, 0
delta = 1e-3  # Small perturbation to x0

ic1 = [x0, y0, z0]
ic2 = [x0+delta, y0, z0]  # Perturbed initial condition

# Solve the system for both initial conditions
sol1 = solve_ivp(lu_chen, t_span, ic1, args=(a, b, c, u), t_eval=t_eval)
sol2 = solve_ivp(lu_chen, t_span, ic2, args=(a, b, c, u), t_eval=t_eval)

t = sol1.t
x1, y1, z1 = sol1.y
x2, y2, z2 = sol2.y

# Create the figure with a 3x6 grid layout
fig = plt.figure(figsize=(12, 8))
gs = fig.add_gridspec(3, 6)  # GridSpec for custom layout

# 3D Phase Space Plot (Occupies 3 rows and 3 columns)
```

```
ax1 = fig.add_subplot(gs[0:3, 0:3], projection='3d')
ax1.plot(x1, y1, z1, label="Trajectory 1 (IC1)", color='blue', alpha=0.7)
ax1.plot(x2, y2, z2, label="Trajectory 2 (IC2)", color='red', linestyle='dashed',
↪  alpha=0.7)
ax1.set_xlabel("X")
ax1.set_ylabel("Y")
ax1.set_zlabel("Z")
ax1.set_title("Lu-Chen System: Phase Space")
ax1.legend()

# Time series for X (Row 0, Columns 3-5)
ax2 = fig.add_subplot(gs[0, 3:6])
ax2.plot(t, x1, label="x1 (IC1)", color='blue')
ax2.plot(t, x2, label="x2 (IC2)", color='red', linestyle='dashed')
ax2.set_xlabel("Time")
ax2.set_ylabel("X")
ax2.set_title("Time Series: X")
ax2.legend()

# Time series for Y (Row 1, Columns 3-5)
ax3 = fig.add_subplot(gs[1, 3:6])
ax3.plot(t, y1, label="y1 (IC1)", color='green')
ax3.plot(t, y2, label="y2 (IC2)", color='orange', linestyle='dashed')
ax3.set_xlabel("Time")
ax3.set_ylabel("Y")
ax3.set_title("Time Series: Y")
ax3.legend()

# Time series for Z (Row 2, Columns 3-5)
ax4 = fig.add_subplot(gs[2, 3:6])
ax4.plot(t, z1, label="z1 (IC1)", color='purple')
ax4.plot(t, z2, label="z2 (IC2)", color='brown', linestyle='dashed')
ax4.set_xlabel("Time")
ax4.set_ylabel("Z")
ax4.set_title("Time Series: Z")
ax4.legend()

plt.tight_layout()
plt.show()
```



We observe from the time series plots that, although the two trajectories start off from nearby initial conditions, their difference grows over time and this becomes particularly noticeable after t = 3.5 for x and y, and after t = 3 for z, indicating that the system is sensitive to initial conditions.

2. Lyapunov Spectrum and Chaos

By computing the Lyapunov exponents, we can determine if the system is chaotic. My code below is again based on lecture 7:

```
from tqdm import tqdm  # Progress tracking

# Lu-Chen system
def lu_chen(t, X, params):
    x, y, z = X[:3]
    Y = X[3:].reshape(3, 3).T
    a, b, c, u = params
    f = np.zeros(12)
    f[:3] = [-a * x + a * y, -x * z + c * y + x + u, x * y - b * z]
    Jac = np.array([[-a, a, 0],
                    [1-z, c, -x],
                    [y, x, -b]])
    f[3:] = (Jac @ Y).T.flatten()
    return f

# Gram-Schmidt reorthogonalisation function
def gram_schmidt(vectors):
    dim = vectors.shape[1]
    ortho_vectors = np.copy(vectors)
```

```python
    norms = np.zeros(dim)

    for i in range(dim):
        for j in range(i):
            proj = np.dot(ortho_vectors[:, j], ortho_vectors[:, i]) * ortho_vectors[:, j]
            ortho_vectors[:, i] -= proj
        norms[i] = np.linalg.norm(ortho_vectors[:, i])
        ortho_vectors[:, i] /= norms[i]

    return ortho_vectors, norms

# Lyapunov spectrum calculation
def lyap_exp(f, dim, params, t_span, t_step, dt, x_0, transient=100):
    t_start, t_end = t_span
    timesteps = int(round((t_end - t_start) / t_step))
    y = np.hstack((x_0, np.eye(dim).flatten()))  # State + tangent vectors
    cum = np.zeros(dim)
    t = t_start

    # Integration and reorthogonalization loop with tqdm progress bar
    for _ in tqdm(range(timesteps), desc="Computing Lyapunov Exponents"):
        sol = solve_ivp(f, [t, t + t_step], y, args=(params,), max_step=dt)
        y = sol.y[:, -1]

        # Extract tangent vectors and reorthogonalize using Gram-Schmidt
        tangent_vectors = y[dim:].reshape(dim, dim).T
        ortho_vectors, norms = gram_schmidt(tangent_vectors)
        y[dim:] = ortho_vectors.T.flatten()

        # Accumulate logarithms of norms
        if t > transient:
            cum += np.log(norms)

        t += t_step

    # Return average Lyapunov exponents
    return cum / (t_end - t_start - transient)


# Set parameters
a, b, c, u = 25.90, 2.98, 21.30, -15.28  # Coefficients
ic = [0, 0, 0]  # Initial condition
t_span = (0, 100)  # Time span for integration
t_step = 0.1  # Integration step size
dt = 0.001  # Maximum step size
transient = 10 # Transient time to discard

# Storage for Lyapunov spectra
lyapunov_spectra = []

params = (a, b, c, u)
L = lyap_exp(lu_chen, dim=3, t_span=t_span, t_step=t_step, dt=dt, x_0=ic, params=params, transient=transient)
lyapunov_spectra.append(L)
```

```python

# Convert results to a numpy array for easy manipulation
lyapunov_spectra = np.array(lyapunov_spectra)

print(lyapunov_spectra)
```

(a), (b) My Lyapunov exponents are 0.80786958, -0.01010516 and -8.36934219. The largest Lyapunov exponent (LLE) is positive, confirming chaos because as seen in lectures, a system is chaotic if and only if its LLE > 0. A Lyapunov exponent measures the exponential rate of divergence or convergence of nearby trajectories in a dynamical system. In particular, for an infinitesimal perturbation δx_0 at time $t = 0$, the distance between two initially close trajectories evolves as

$$|\delta x(t)| \approx |\delta x_0| e^{\lambda t}$$

where λ is the Lyapunov exponent. This means that if $\lambda > 0$, small perturbations grow exponentially, leading to sensitivity to initial conditions and chaos.

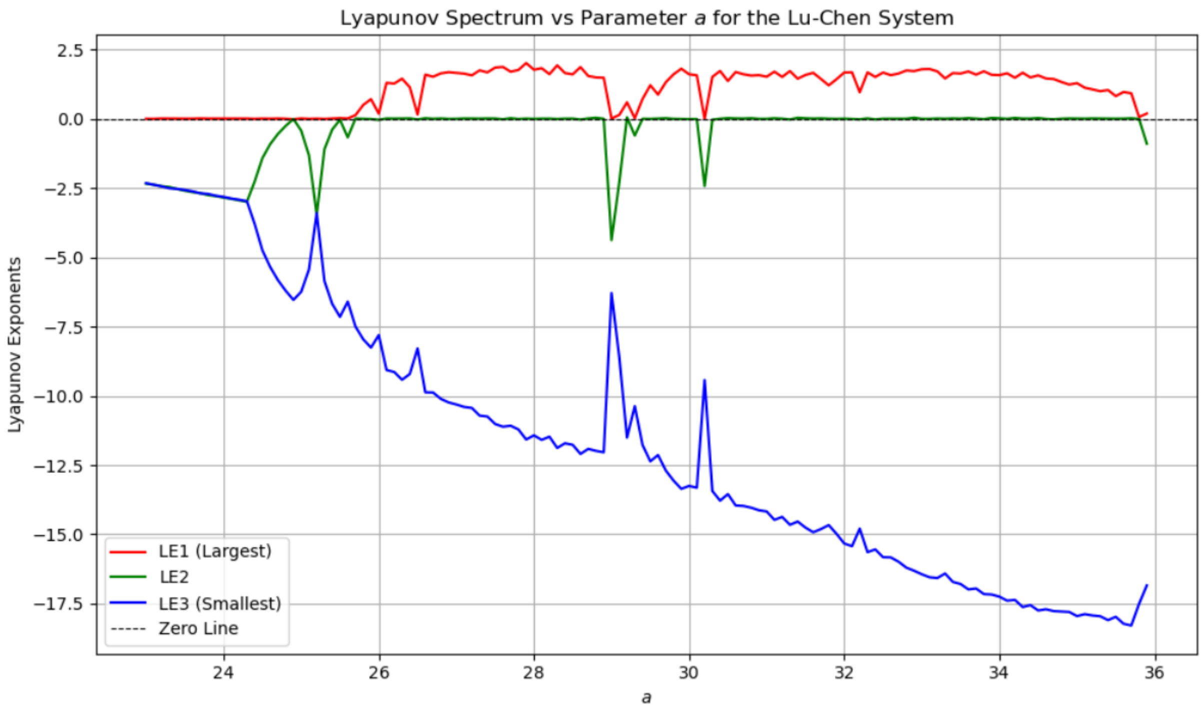
The explanation above is consistent with our time series plots, which shows the trajectories diverging over time, despite starting from nearby initial conditions. The second exponent is close to zero, representing neutral motion along the attractor. The third exponent is negative, which indicates contraction to the attractor.

Code for parameter sweep:

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp
from tqdm import tqdm  # Progress tracking

# Gram-Schmidt reorthogonalisation function
def gram_schmidt(vectors):
    dim = vectors.shape[1]
    ortho_vectors = np.copy(vectors)
    norms = np.zeros(dim)

    for i in range(dim):
        for j in range(i):
            proj = np.dot(ortho_vectors[:, j], ortho_vectors[:, i]) * ortho_vectors[:, j]
            ortho_vectors[:, i] -= proj
        norms[i] = np.linalg.norm(ortho_vectors[:, i])
        ortho_vectors[:, i] /= norms[i]

    return ortho_vectors, norms

# Lyapunov spectrum calculation
def lyap_exp(f, dim, params, t_span, t_step, dt, x_0, transient=10):
    t_start, t_end = t_span
    timesteps = int(round((t_end - t_start) / t_step))
    y = np.hstack((x_0, np.eye(dim).flatten()))  # State + tangent vectors
    cum = np.zeros(dim)
    t = t_start

    # Integration and reorthogonalisation loop
    for _ in range(timesteps):
        sol = solve_ivp(f, [t, t + t_step], y, args=(params,), max_step=dt)
        y = sol.y[:, -1]
```

```
        # Extract tangent vectors and reorthogonalise using Gram-Schmidt
        tangent_vectors = y[dim:].reshape(dim, dim).T
        ortho_vectors, norms = gram_schmidt(tangent_vectors)
        y[dim:] = ortho_vectors.T.flatten()

        # Accumulate logarithms of norms
        if t > transient:
            cum += np.log(norms)

        t += t_step

    # Return average Lyapunov exponents
    return cum / (t_end - t_start - transient)

# Lu-Chen system
def lu_chen(t, X, params):
    x, y, z = X[:3]
    Y = X[3:].reshape(3, 3).T
    a, b, c, u = params
    f = np.zeros(12)
    f[:3] = [-a * x + a * y, -x * z + c * y + x + u, x * y - b * z]
    Jac = np.array([[-a, a, 0],
                    [1-z, c, -x],
                    [y, x, -b]])
    f[3:] = (Jac @ Y).T.flatten()
    return f

# Sweep parameter `a`
a_values = np.arange(23, 36, 0.25)  ########## step size?
a, b, c, u = 25.90, 2.98, 21.30, -15.28  # Coefficients
ic = [0, 0, 0]  # Initial condition
t_span = (0, 100)  # Time span for integration
t_step = 0.1  # Integration step size
dt = 0.001  # Maximum step size
transient = 10 # Transient time to discard

# Storage for Lyapunov spectra
lyapunov_spectra = []

# Sweep `a` and calculate Lyapunov spectra
for a in tqdm(a_values, desc="Sweeping parameter a"):
    params = (a, b, c, u)
    L = lyap_exp(lu_chen, dim=3, t_span=t_span, t_step=t_step, dt=dt, x_0=ic, params=params, transient=transient)
    lyapunov_spectra.append(L)

# Convert results to a numpy array for easy manipulation
lyapunov_spectra = np.array(lyapunov_spectra)

# Plotting the Lyapunov spectrum as a function of `a`
plt.figure(figsize=(10, 6))
plt.plot(a_values, lyapunov_spectra[:, 0], label='LE1 (Largest)', color='r')
plt.plot(a_values, lyapunov_spectra[:, 1], label='LE2', color='g')
```

```python
plt.plot(a_values, lyapunov_spectra[:, 2], label='LE3 (Smallest)', color='b')
plt.axhline(0, color='black', linewidth=0.8, linestyle='--', label='Zero Line')
plt.xlabel('$a$')
plt.ylabel('Lyapunov Exponents')
plt.title('Lyapunov Spectrum vs Parameter $a$ for the Lu-Chen System')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```



(c) Whenever the red curve lies on the y axis (for example, when a = 24), the behaviour is periodic, as the red curve shows the LLE. Whenever the red curve lies above the y axis (for example, when a = 32), the behaviour is chaotic, because as explained earlier, chaos is characterised by LLE > 0. Whenever the green curve is zero, there is neutral motion along the attractor.

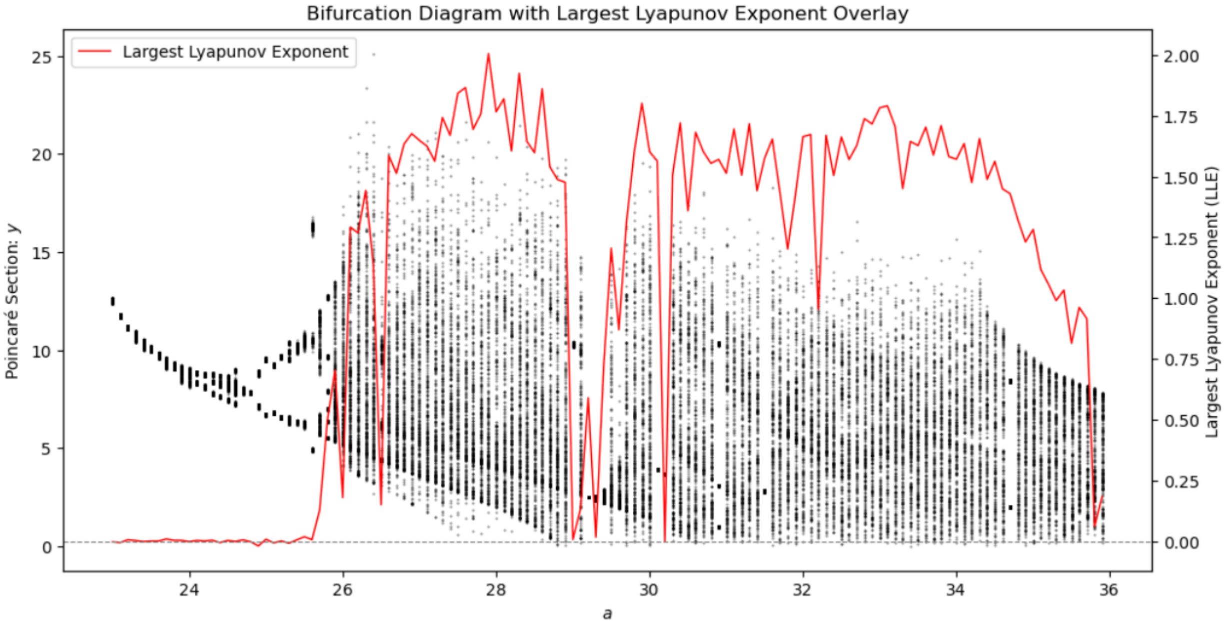
3. Bifurcation Analysis via Poincare Sections

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp
from tqdm import tqdm  # Progress tracking

# Define the Lu-Chen system
def lu_chen(t, state, a, b, c, u):
    x, y, z = state
    dxdt = -a * x + a * y
    dydt = -x * z + c * y + x + u
    dzdt = x * y - b * z
    return [dxdt, dydt, dzdt]

# Parameters
```

```python
b, c, u = 2.98, 21.30, -15.28  # Fixed coefficients
ic = [0, 0, 0]  # Initial condition

def poincare_section(a, T=100, dt=0.001, transient=80):
    """
    Computes the Poincaré section of the Lu-Chen system for a given a, discarding transient states.
    Uses linear interpolation to find more accurate crossing points.
    """
    t_eval = np.arange(0, T, dt)
    sol = solve_ivp(lu_chen, [0, T], ic, args=(a, b, c, u), t_eval=t_eval, method='RK45')
    x_vals, y_vals, z_vals = sol.y

    # Identify indices beyond transient time
    transient_idx = np.searchsorted(t_eval, transient)

    # Extract Poincaré section using linear interpolation
    poincare_y = []
    for i in range(transient_idx, len(x_vals) - 1):
        if x_vals[i-1] < 0 and x_vals[i] > 0:  # Crossing x = 0 with dx/dt > 0
            # Linear interpolation for better accuracy
            x1, x2 = x_vals[i-1], x_vals[i]
            y1, y2 = y_vals[i-1], y_vals[i]
            x0_frac = -x1 / (x2 - x1)  # Fraction of step where x = 0
            y_interp = y1 + x0_frac * (y2 - y1)  # Interpolated y value
            poincare_y.append(y_interp)

    return poincare_y

# Sweep a and collect bifurcation data
a_values = np.arange(23, 36, 0.1)  # Finer resolution
bifurcation_data = []

for a in tqdm(a_values, desc="Computing bifurcation diagram"):
    y_poincare = poincare_section(a)
    bifurcation_data.extend([(a, y) for y in y_poincare])

bifurcation_data = np.array(bifurcation_data)  # Convert to numpy array

# Plot bifurcation diagram
fig, ax1 = plt.subplots(figsize=(12, 6))

# Scatter plot for bifurcation diagram
ax1.scatter(bifurcation_data[:, 0], bifurcation_data[:, 1], s=0.2, color="black", alpha=0.5)
ax1.set_xlabel("$a$")
ax1.set_ylabel("Poincaré Section: $y$")
ax1.set_title("Bifurcation Diagram with Largest Lyapunov Exponent Overlay")

# Secondary y-axis for LLE
ax2 = ax1.twinx()
ax2.plot(a_values, lyapunov_spectra[:, 0], label="Largest Lyapunov Exponent", color="red", linewidth=1)
ax2.set_ylabel("Largest Lyapunov Exponent (LLE)")
```

```
ax2.axhline(0, color="gray", linestyle="dashed", linewidth=0.8)  # Mark LLE = 0

# Add legend for LLE
ax2.legend(loc="upper left")

plt.show()
```

Old graph (before linear interpolation was added):



We see from the above plot that whenever the largest Lyapunov exponent (LLE) is positive, the (long-term) behaviour is chaotic, but when the LLE is zero, the behaviour is periodic. We see a period-doubling cascade from period 1 to period 2 around a = 25, and another one when a = 25.9 (approximately), transitioning from period 2 to period 4. Beyond a = 26, the behaviour becomes chaotic, with some exceptions, such as a = 29.2 and 30.2. Period doubling cascades like the one shown here is a universal route to chaos, as seen in lecture 10.

Part 2: Synchronisation and Data-Driven Analysis

1. Secure Communication via Chaotic Synchronisation

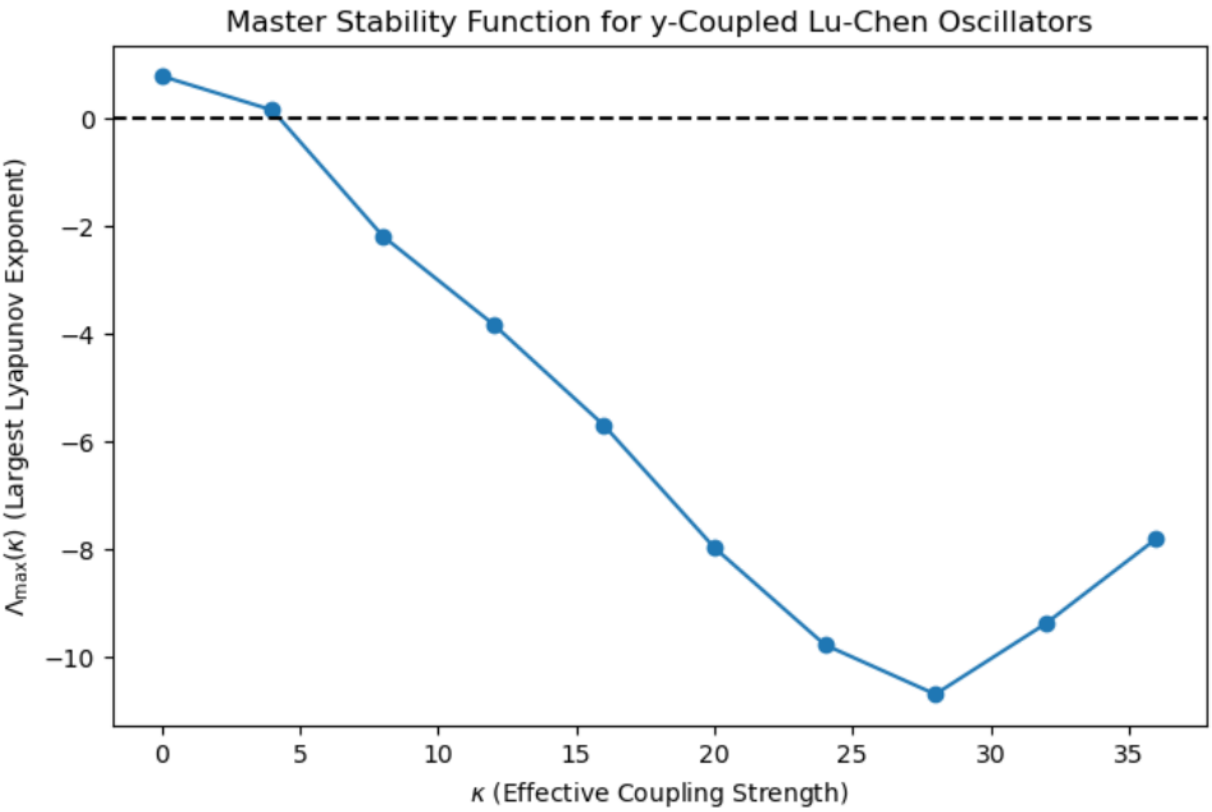
My code is based on lecture 12: Synchronization of Chaos.

Code for Master Stability Function (MSF):

```
import numpy as np
from scipy.integrate import solve_ivp
from tqdm import tqdm
import matplotlib.pyplot as plt

# Gram-Schmidt reorthogonalisation function
def gram_schmidt(vectors):
    dim = vectors.shape[1]
    ortho_vectors = np.copy(vectors)
    norms = np.zeros(dim)
```

```python

    for i in range(dim):
        for j in range(i):
            proj = np.dot(ortho_vectors[:, j], ortho_vectors[:, i]) * ortho_vectors[:, j]
            ortho_vectors[:, i] -= proj
        norms[i] = np.linalg.norm(ortho_vectors[:, i])
        ortho_vectors[:, i] /= norms[i]

    return ortho_vectors, norms

# Lyapunov exponent calculation
def lyapunov_exponent(f, dim, params, t_span, t_step, dt, x_0, transient=100):
    t_start, t_end = t_span
    timesteps = int(round((t_end - t_start) / t_step))
    y = np.hstack((x_0, np.eye(dim).flatten()))  # State + tangent vectors
    cum = np.zeros(dim)
    t = t_start

    # Integration and reorthogonalisation loop
    for _ in range(timesteps):
        sol = solve_ivp(f, [t, t + t_step], y, args=(params,), max_step=dt)  # Pass
        ↪ params as tuple
        y = sol.y[:, -1]

        # Extract tangent vectors and reorthogonalise using Gram-Schmidt
        tangent_vectors = y[dim:].reshape(dim, dim).T
        ortho_vectors, norms = gram_schmidt(tangent_vectors)
        y[dim:] = ortho_vectors.T.flatten()

        # Accumulate logarithms of norms
        if t > transient:
            cum += np.log(norms)

        t += t_step

    return cum / (t_end - t_start - transient)

# General Master Stability Function computation
def master_stability_function(system, dim, params, kappa_values, t_span, t_step, dt, x_0,
↪ transient=100):
    msf_values = []

    for kappa in tqdm(kappa_values, desc="Computing MSF"):
        full_params = params + (kappa,)  # Ensure kappa is included in params tuple
        L = lyapunov_exponent(system, dim=dim, params=full_params,
                         t_span=t_span, t_step=t_step, dt=dt, x_0=x_0, transient=transient)
        msf_values.append(L[0])  # Only consider the largest Lyapunov exponent

    return np.array(msf_values)

# Define the Lu-Chen system equations
def lu_chen_system(t, x, a, b, c, u):
    return np.array([-a*x[0]+a*x[1], -x[0]*x[2] + c*x[1] + x[0] + u, x[0]*x[1] - b*x[2]])

```

```python
# Variational equation for the Master Stability Function
def msf_variational_eq(t, x, params):
    a, b, c, u, kappa = params  # Unpacking params correctly
    Y = x[3:].reshape(3, 3).T  # Reshape tangent vectors
    f = np.zeros(12)
    f[:3] = lu_chen_system(t, x[:3], a, b, c, u)  # Compute Lu-Chen dynamics

    # Jacobian matrix with kappa-dependent coupling
    J = np.array([[-a, a, 0],
                  [1-x[2], c-kappa, -x[0]],
                  [x[1], x[0], -b]])

    f[3:] = (J @ Y).T.flatten()  # Apply Jacobian to tangent vectors
    return f

# Define parameters for the Lu-Chen system
a, b, c, u = 25.90, 2.98, 21.30, -15.28
params = (a, b, c, u)  # FIXED: Only pass a, b, c, u here (kappa is added later)
x_0 = np.array([1.0, 1.0, 1.0])  # Initial condition
t_span = (0, 1000)  # Time span
t_step = 0.1  # Integration step size
dt = 0.01  # Maximum step size
transient = 100  # Transient time

# Define the range of kappa values
kappa_values = np.arange(0, 40, 4)

# Compute the Master Stability Function
msf_results = master_stability_function(msf_variational_eq, dim=3, params=params, kappa_values=kappa_values,
                                        t_span=t_span, t_step=t_step, dt=dt, x_0=x_0, transient=transient)

# Plot the MSF
plt.figure(figsize=(8,5))
plt.plot(kappa_values, msf_results, marker='o', linestyle='-')
plt.axhline(0, color='k', linestyle='--')
plt.xlabel(r'$\kappa$ (Effective Coupling Strength)')
plt.ylabel(r'$\Lambda_{\max}(\kappa)$ (Largest Lyapunov Exponent)')
plt.title("Master Stability Function for y-Coupled Lu-Chen Oscillators")
plt.show()
```



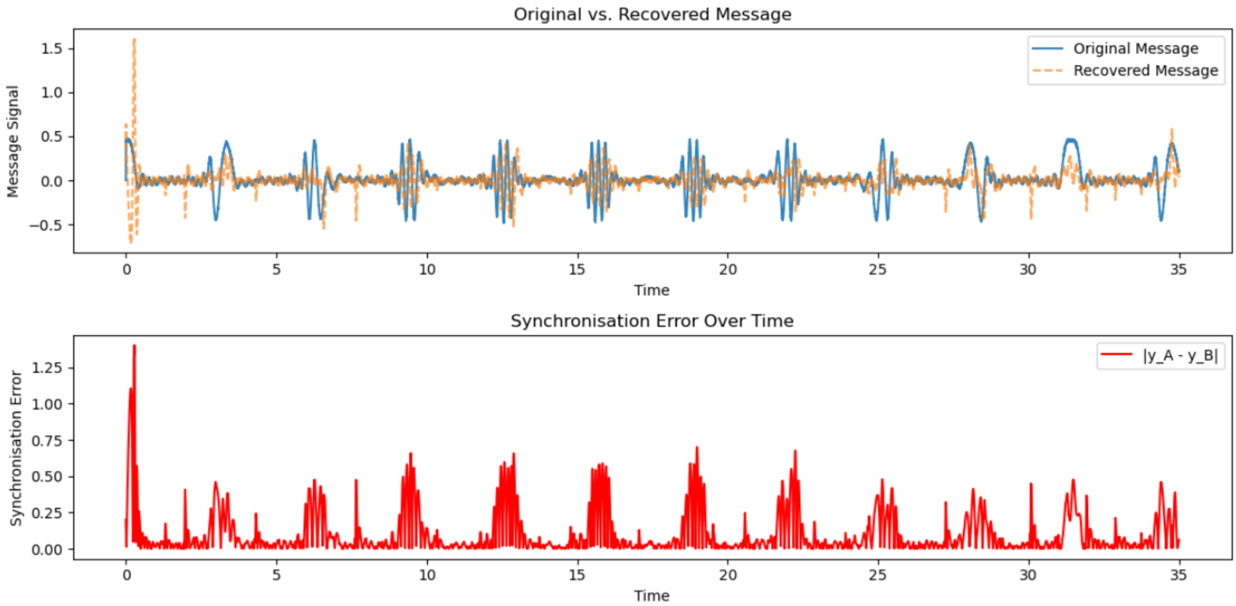
Since chaos is characterised by the condition $\Lambda_{max} > 0$ (in which case Bob's and Alice's systems would not be able to synchronise), we need to choose a coupling strength κ such that $\Lambda_{max}(\kappa) < 0$. As shown in the plot above, $\kappa = 28$ satisfies this.

Code for synchronisation:

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp

# Lu-Chen system parameters
a, b, c, u = 25.90, 2.98, 21.30, -15.28
alpha = 28  # Coupling constant

# Define the Lu-Chen system (Alice)
def lu_chen(t, state, a, b, c, u):
    x, y, z = state
    dxdt = -a * x + a * y
    dydt = -x * z + c * y + x + u
    dzdt = x * y - b * z
    return [dxdt, dydt, dzdt]

# Define the message signal
def message_signal(t):
    noise = np.random.normal(0, 0.01, len(t))  # Small Gaussian noise
    return 0.1 * np.sin((1.2 * np.pi * np.sin(t))**2) / (np.pi * np.sin(t)**2 + 1e-9) *
    ↪  np.cos(10 * np.pi * np.cos(0.9*t)) + noise

```

```python
# Time span
t_span = (0, 35)
t_eval = np.linspace(t_span[0], t_span[1], 10000)  # High resolution

# Alice's initial conditions
state0_Alice = x_0

# Solve Alice's system
sol_Alice = solve_ivp(lu_chen, t_span, state0_Alice, args=(a, b, c, u), t_eval=t_eval)
x_Alice, y_Alice, z_Alice = sol_Alice.y[:3]  # Extract x, y, z components

# Generate the message and transmit
m_t = message_signal(t_eval)
y_transmit = y_Alice + m_t


# Define Bob's synchronised Lu-Chen system with a coupling term
def lu_chen_bob(t, state, a, b, c, u, alpha, t_eval, y_Alice, m_t):
    x2, y2, z2 = state

    s = np.interp(t, t_eval, y_transmit)

    # Bob's system equations
    dx2dt = -a * x2 + a * y2
    dy2dt = -x2 * z2 + c * y2 + x2 + u + alpha * (s - y2)   # coupling
    dz2dt = x2 * y2 - b * z2

    return [dx2dt, dy2dt, dz2dt]

# Initial conditions for Bob
state0_Bob = [0.8, 0.8, 0.8]

# Solve Bob's system
sol_Bob = solve_ivp(lu_chen_bob, t_span, state0_Bob, args=(a, b, c, u, alpha, t_eval, y_Alice, m_t), t_eval=t_eval)
y_Bob = sol_Bob.y[1]

# Recover the message
m_recovered = y_transmit - y_Bob

# Compute synchronisation error
sync_error = np.abs(y_Alice - y_Bob)

# Plot results
plt.figure(figsize=(12, 6))

# Original vs. Recovered message
plt.subplot(2, 1, 1)
plt.plot(t_eval, m_t, label="Original Message", alpha=0.9)
plt.plot(t_eval, m_recovered, label="Recovered Message", linestyle="dashed", alpha=0.7)
plt.xlabel("Time")
plt.ylabel("Message Signal")
plt.legend()
plt.title("Original vs. Recovered Message")
```

```
# Synchronisation error
plt.subplot(2, 1, 2)
plt.plot(t_eval, sync_error, label="|y_A - y_B|", color="red")
plt.xlabel("Time")
plt.ylabel("Synchronisation Error")
plt.legend()
plt.title("Synchronisation Error Over Time")

plt.tight_layout()
plt.show()
```



We can see that, apart from the initially high synchronisation error (as synchronisation does not happen immediately), the error is roughly 0.5 - 0.75, but considering that the the magnitude of the message signal is only 0.5, the relative error is quite high. One possible reason is solving the differential equations numerically with solve_ivp. Additionally, the system is chaotic, which can make synchronisation difficult, and we have introduced some noise in the message, so a fully accurate recovery is not possible.

2. Network Reconstruction

My method for recovering the network is based on the reduction theorem from lecture 20. The process consists of four parts:

1. Classifying nodes based on in-degree.
2. Learning local dynamics $\mathbf{f}$ from low-degree nodes.
3. Extracting the coupling function $\mathbf{H}$ from hubs.
4. Recovering the connectivity matrix $\mathbf{L}$ via sparse regression.

I used SINDy to estimate the internal dynamics because Ridge, Lasso and Compressed Sensing do not explicitly consider system dynamics. My function library consists of polynomials up to degree 2 because even if I include any cubic terms, SINDy sets the cubic coefficients to zero. Indeed, any nonzero cubic (or higher order) coefficient would be very small and most likely arise from noise. My code for SINDy and the function library is based on lecture 18:

```python
import numpy as np
import pandas as pd
from tqdm import tqdm  # Progress bar

# Load data
data = pd.read_csv("network_dynamics_data.txt", delimiter=" ")

times = data.iloc[:, 0].values  # Time column
all_streams = data.iloc[:, 1:].values  # State variables

# Reshape data into a dictionary {node_id: time_series_data}
num_nodes = 5
node_vars = 3  # (x, y, z)
time_series = {i + 1: all_streams[:, i * node_vars : (i + 1) * node_vars] for i in range(num_nodes)}

# Function to construct feature library
def build_library(X, node_id):
    """Constructs a basis of functions including quadratic terms for a specific node."""
    n_samples, n_features = X.shape
    library = [np.ones(n_samples)]  # Constant term
    terms = ["1"]  # Labels for terms

    var_names = ["x", "y", "z"]  # Labels for variables
    for i in range(n_features):
        library.append(X[:, i])
        terms.append(f"{var_names[i]}{node_id}")  # linear terms

    for i in range(n_features):
        for j in range(i, n_features):  # i to avoid duplicate terms
            library.append(X[:, i] * X[:, j])
            if i == j:
                terms.append(f"{var_names[i]}{node_id}**2")  # self quadratic terms
            else:
                terms.append(f"{var_names[i]}{node_id} * {var_names[j]}{node_id}")  # cross quadratic terms

    return np.column_stack(library), terms


threshold = 0.5  # threshold for SINDy

# Sparse regression function
def sindy(A, dXdt):
    """Performs sparse regression using iterative thresholding."""
    coeffs = np.zeros((A.shape[1], dXdt.shape[1]))
    for i in tqdm(range(dXdt.shape[1]), desc="Fitting SINDy models"):
        x = np.linalg.pinv(A) @ dXdt[:, i]
        for _ in range(15):
            small = np.abs(x) < threshold
            x[small] = 0
            if np.any(~small):
                x[~small] = np.linalg.pinv(A[:, ~small]) @ dXdt[:, i]
        coeffs[:, i] = x
```

```
    return coeffs

# Function to display differential equations
def print_equations(coeffs, terms_dict):
    """Formats and prints differential equations with correct variable labels."""
    for node, coef_matrix in coeffs.items():
        print(f"\nDifferential equations for Node {node}:")
        terms = terms_dict[node]
        for var_idx, var in enumerate(["x", "y", "z"]):
            equation_terms = [
                f"{coef:.3f} * {term}" for coef, term in zip(coef_matrix[:, var_idx],
                ↪  terms) if abs(coef) > 1e-3
            ]
            equation = " + ".join(equation_terms) if equation_terms else "0"
            print(f"d{var}{node}/dt = {equation}")

# Infer SINDy coefficients
def infer_sindy_coefficients(time_series):
    coeffs = {}
    terms_dict = {}
    for node, X in time_series.items():
        X_t, X_t1 = X[:-1], X[1:]  # Shifted time series
        dt = np.mean(np.diff(times))  # Time step
        X_dot = (X_t1 - X_t) / dt  # Finite difference approximation

        A, terms = build_library(X_t, node)
        coeffs[node] = sindy(A, X_dot)
        terms_dict[node] = terms

    return coeffs, terms_dict

# Compute and display results
coeffs, terms_dict = infer_sindy_coefficients(time_series)
print_equations(coeffs, terms_dict)
```

Code output:

```
Differential equations for Node 1:
dx1/dt = -26.330 * x1 + 24.839 * y1
dy1/dt = -15.116 * 1 + 0.724 * x1 + 19.930 * y1 + -0.985 * x1 * z1
dz1/dt = 1.986 * 1 + -3.104 * z1 + 0.999 * x1 * y1

Differential equations for Node 2:
dx2/dt = -25.516 * x2 + 25.008 * y2
dy2/dt = -14.752 * 1 + 20.152 * y2 + -0.969 * x2 * z2
dz2/dt = 1.422 * 1 + -3.069 * z2 + 0.996 * x2 * y2

Differential equations for Node 3:
dx3/dt = -25.399 * x3 + 24.877 * y3
dy3/dt = -14.848 * 1 + 20.094 * y3 + -0.967 * x3 * z3
dz3/dt = 1.498 * 1 + -3.083 * z3 + 0.998 * x3 * y3

Differential equations for Node 4:
dx4/dt = -26.143 * x4 + 25.047 * y4
dy4/dt = -15.055 * 1 + 0.547 * x4 + 19.878 * y4 + -0.976 * x4 * z4
dz4/dt = 2.545 * 1 + -3.133 * z4 + 0.999 * x4 * y4

Differential equations for Node 5:
dx5/dt = -25.601 * x5 + 24.985 * y5
dy5/dt = -14.862 * 1 + 20.112 * y5 + -0.967 * x5 * z5
dz5/dt = 1.951 * 1 + -3.102 * z5 + 0.999 * x5 * y5
```

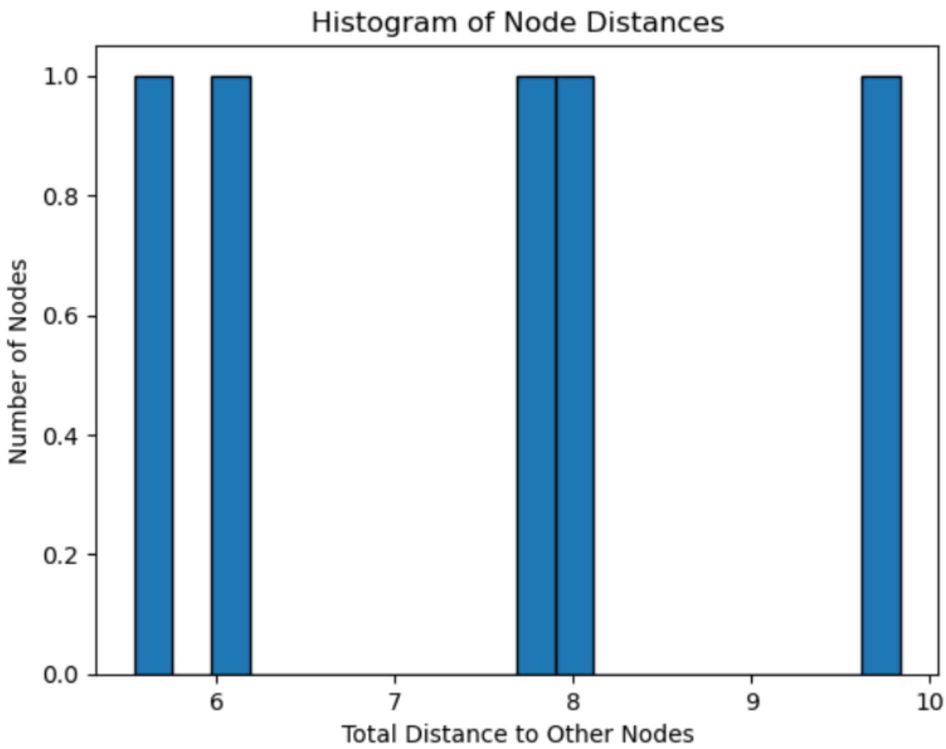
We will now identify hub nodes by means of a distance matrix to measure the similarity between inferred models, as seen in lecture 20. The idea is that low degree nodes will cluster together as they roughly follow the same isolated dynamics. On the other hand, hub nodes will be distinct due to greater coupling effects. The distance between two systems i and j is defined as

$$d_{ij} = \left(\sum_{k=1}^{p} \frac{1}{V_k} |\xi_i^k - \xi_j^k|^2 \right)^{1/2},$$

where:
- ξ_i^k are the inferred regression coefficients for node i,
- V_k is the variance of the predicted coefficients for basis function k. The corresponding code is shown below. We will compute the row-sum of the distance matrix, from which we obtain a histogram.

```python
import numpy as np
import matplotlib.pyplot as plt
import re

# Extract generalised basis function names
sample_node = next(iter(terms_dict.values()))  # Get any node's basis function list

basis_functions = [re.sub(r'([xyz])\d+', r'\1', term) for term in sample_node]    # ['1',
↪  'x', 'y', 'z', 'x**2', 'x * y', 'x * z', 'y**2', 'y * z', 'z**2']


num_basis_functions = len(basis_functions)

```

```python
# Sort nodes
node_ids = sorted(coeffs.keys())
num_nodes = len(node_ids)

# Create coefficient matrix (num_nodes, num_basis_functions)
coeff_matrix = np.zeros((num_nodes, num_basis_functions))  # shape is (5, 10)


for i, node in enumerate(node_ids):
    for j, term in enumerate(terms_dict[node]):
        generalised_term = term.replace(f"x{node}", "x").replace(f"y{node}", "y").replace(f"z{node}", "z")
        if generalised_term in basis_functions:
            coeff_matrix[i, basis_functions.index(generalised_term)] = coeffs[node][j][0]


# Compute variance per basis function across nodes
V_k = np.var(coeff_matrix, axis=0, ddof=1)
V_k[V_k == 0] = 0.000001  # Avoid division by zero

# Compute pairwise distance matrix
distance_matrix = np.zeros((num_nodes, num_nodes))
for i in range(num_nodes):
    for j in range(i + 1, num_nodes):
        diff = coeff_matrix[i] - coeff_matrix[j]
        distance_matrix[i, j] = np.sqrt(np.sum((diff ** 2) / V_k))  # Variance-scaled distance
        distance_matrix[j, i] = distance_matrix[i, j]  # Symmetric matrix

# Compute node importance
node_importance = np.sum(distance_matrix, axis=1)

# Plot histogram
plt.hist(node_importance, bins=20, edgecolor="black")
plt.xlabel("Total Distance to Other Nodes")
plt.ylabel("Number of Nodes")
plt.title("Histogram of Node Distances")
plt.show()

# Print node importance scores
for node, importance in zip(node_ids, node_importance):
    print(f"Node {node}: Total Distance = {importance:.3f}")
```




```
Node 1: Total Distance = 9.833
Node 2: Total Distance = 6.147
Node 3: Total Distance = 7.744
Node 4: Total Distance = 8.082
Node 5: Total Distance = 5.540
```

Based off the histogram, nodes 1, 2 and 4 are likely to be hubs, as these have the highest row-sums. We will shortly verify that 1 and 4 are indeed hubs, while nodes 2, 3 and 5 are low degree nodes. We will therefore take the average of the latter three nodes to compute the local dynamics:

```python
import numpy as np
import re

threshold = 0.1

def local_dynamics_equations(f_local, terms):
    """Prints local dynamics equations in the form f_x local = ... up to f_z local =
    ..."""
    for var_idx, var in enumerate(["x", "y", "z"]):
        eq_terms = []
        for coef, term in zip(f_local[:, var_idx], terms):
            if abs(coef) > threshold:
                # Replace any indexed variable (e.g., x1, z2) with its general form
                term = re.sub(r"[xyz]\d+", lambda m: m.group(0)[0], term)
                eq_terms.append(f"{coef:.3f} * {term}")

        equation = " + ".join(eq_terms) if eq_terms else "0"
        print(f"f_{var} local = {equation}")


# Compute f_local, filtering out hub nodes first
```

```python
non_hub_coeffs = [v for k, v in coeffs.items() if k not in hub_nodes]
f_local = np.mean(np.array(non_hub_coeffs), axis=0)

# Print local dynamics
f_local_values = local_dynamics_equations(f_local, terms_dict[1])
```

The results are given below:

```
f_x local = -25.505 * x + 24.956 * y
f_y local = -14.821 * 1 + 20.119 * y + -0.967 * x * z
f_z local = 1.624 * 1 + -3.084 * z + 0.998 * x * y
```

As seen in lecture 20, under the mean-field approximation, the discrete-time dynamics can be approximated as

$$\mathbf{x}_h(t+1) - \mathbf{x}_h(t) \approx \mathbf{f}(\mathbf{x}_h) + k_h \mathbf{V}(\mathbf{x}_h) + \mathbf{C}$$

where $\mathbf{x} = (x, y, z)$, $\mathbf{f}(\mathbf{x}_h)$ is the isolated dynamics of hub node h, k_h is the in-degree of h, $V(\mathbf{x}_h)$ is the effective coupling function for node h, and $\mathbf{C} = (C_1, C_2, C_3)$ is an integration constant arising from the mean field approximation. We will now generalise the difference equation above with $\Delta t = 1$ to any time increment Δt, because in our dataset, $\Delta t \approx 0.0001$. We start by rewriting the discrete-time equation as:

$$\mathbf{x}_h(t + \Delta t) - \mathbf{x}_h(t) \approx \Delta t\{\mathbf{f}(\mathbf{x}_h) + k_h \mathbf{V}(\mathbf{x}_h) + \mathbf{C}\}$$

with $\Delta t = 1$. Dividing both sides by Δt and rearranging, we obtain

$$\frac{\mathbf{x}_h(t + \Delta t) - \mathbf{x}_h(t)}{\Delta t} - \mathbf{f}(\mathbf{x}_h) \approx k_h \mathbf{V}(\mathbf{x}_h) + \mathbf{C}$$

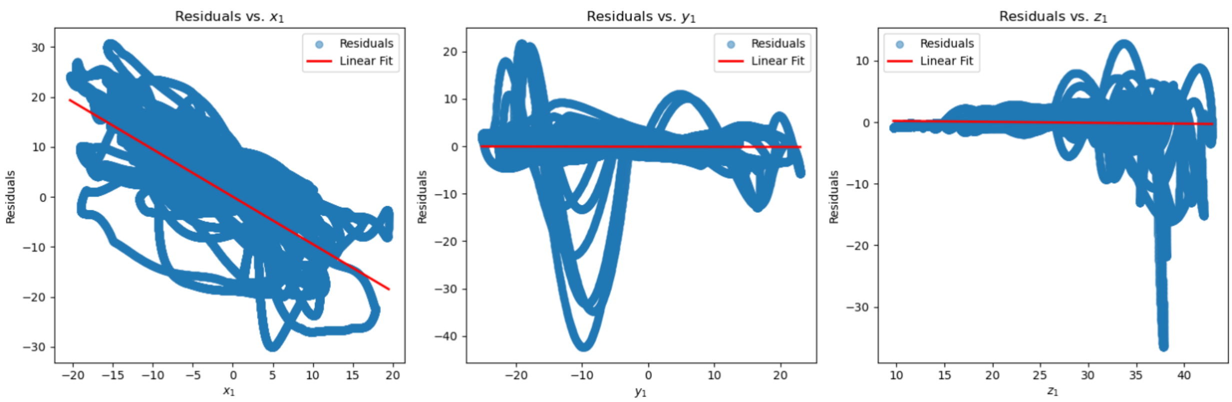
For each t, I decided to plot $k_h \mathbf{V}(\mathbf{x}_h) + \mathbf{C}$ (by computing the LHS of the above equation) against x, y and z to identify any patterns.

```python
import numpy as np
import sympy as sp


def local_dynamics(f_local, time_series_i, T, basis_functions):
    """
    Computes local dynamics using SymPy for symbolic evaluation.

    Args:
        f_local (np.ndarray): Shape (num_terms, 3), representing local dynamics
        ↪  coefficients.
        terms (list of str): List of term names corresponding to `f_local`.
        time_series_i (np.ndarray): Shape (T, 3), time series for node i.
        T (int): Number of time steps.
        basis_functions (list of str): List of basis function names.

    Returns:
        np.ndarray: Shape (3, T), containing f_x, f_y, f_z values over time.
    """
    # Define symbolic variables
    x, y, z = sp.symbols("x y z")

    # Dynamically create a symbolic dictionary from basis_functions
    term_expressions = {term: eval(term, {"x": x, "y": y, "z": z}) for term in
    ↪  basis_functions}
```

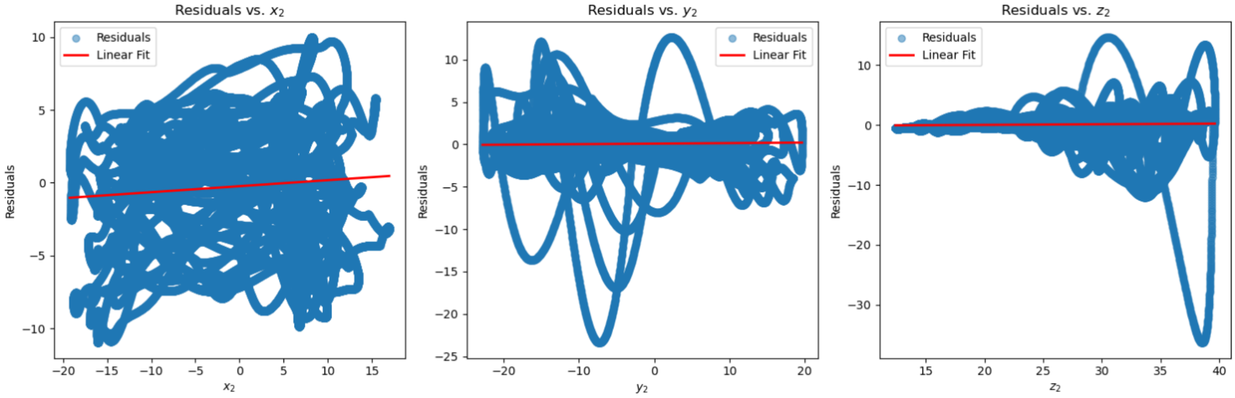
```python

    # Convert f_local into symbolic expressions
    f_x_expr = sum(f_local[j, 0] * term_expressions[basis_functions[j]] for j in
    ↪  range(num_basis_functions))
    f_y_expr = sum(f_local[j, 1] * term_expressions[basis_functions[j]] for j in
    ↪  range(num_basis_functions))
    f_z_expr = sum(f_local[j, 2] * term_expressions[basis_functions[j]] for j in
    ↪  range(num_basis_functions))


    # Convert symbolic expressions into numerical functions
    f_x_func = sp.lambdify((x, y, z), f_x_expr, "numpy")
    f_y_func = sp.lambdify((x, y, z), f_y_expr, "numpy")
    f_z_func = sp.lambdify((x, y, z), f_z_expr, "numpy")

    # Compute local dynamics over time (3, T)
    f_x_values = np.array([f_x_func(*time_series_i[t, :3]) for t in range(T)])
    f_y_values = np.array([f_y_func(*time_series_i[t, :3]) for t in range(T)])
    f_z_values = np.array([f_z_func(*time_series_i[t, :3]) for t in range(T)])

    return np.vstack([f_x_values, f_y_values, f_z_values])  # Shape (3, T)



h = 1  # node number
T = times.shape[0]  # Number of time points to evaluate


# Compute local dynamics
f_local_values_h = local_dynamics(f_local, time_series[h], T, basis_functions)


stream_h = time_series[h]  # Shape (150000, 3)

# Compute time increments dynamically
dt = np.diff(times)  # Shape (T-1,)

# Reshape dt to align with broadcasting requirements
dt = dt[:, None]  # Shape (T-1, 1)

# Compute numerical derivatives using finite differences
stream_h_derivative = np.diff(stream_h, axis=0) / dt  # Shape (149999, 3)

# The matrix below has shape (T-1, 3)
residual_matrix = stream_h_derivative[:, :] - f_local_values_h[:, :-1].T  # k_h V(s_h) +
↪  C where s = (x, y, z)

import numpy as np
import matplotlib.pyplot as plt
import warnings
warnings.simplefilter("ignore")

# Create subplots
fig, axes = plt.subplots(1, 3, figsize=(15, 5))
```

```python

# Use actual value of h in labels
var_names = [f"${v}_{h}$" for v in ["x", "y", "z"]]

# Loop over the three variables
for i in range(3):
    x = time_series[h][:-1, i]  # Independent variable
    y = residual_matrix[:, i]  # Dependent variable

    # Scatter plot
    axes[i].scatter(x, y, alpha=0.5, label="Residuals")

    # Fit a straight line (1st-degree polynomial)
    coeffs = np.polyfit(x, y, 1)  # Returns [slope, intercept]
    poly_eq = np.poly1d(coeffs)  # Create polynomial function

    # Generate fitted values
    x_fit = np.linspace(np.min(x), np.max(x), 100)  # Smooth range for line
    y_fit = poly_eq(x_fit)

    # Plot the fitted line
    axes[i].plot(x_fit, y_fit, color="red", linewidth=2, label="Linear Fit")

    # Labels and title with updated h
    axes[i].set_xlabel(var_names[i])
    axes[i].set_ylabel("Residuals")
    axes[i].set_title(f"Residuals vs. {var_names[i]}")
    axes[i].legend()

    # Print the equation of the fitted line with h replaced
    slope, intercept = coeffs
    base = ["x", "y", "z"][i]  # Extract the variable name

    print(f"Equation for {base}_{h}: res_{base}_{h} = {slope:.4f}{base}_{h} {'+' if
    ↪ intercept >= 0 else '-'} {abs(intercept):.4f}")

plt.tight_layout()
plt.show()
```

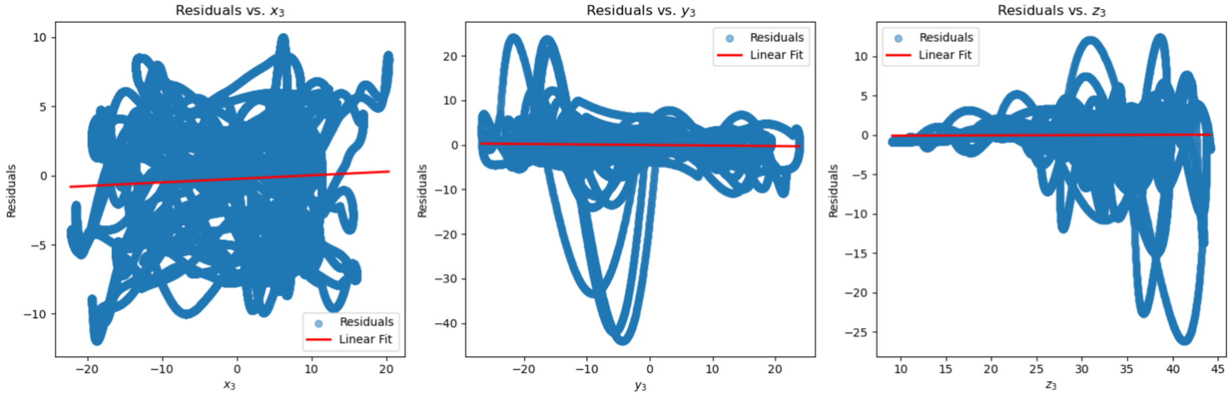
```
Equation for x_1: res_x_1 = -0.9490x_1 + 0.0397
Equation for y_1: res_y_1 = -0.0024y_1 - 0.0916
Equation for z_1: res_z_1 = -0.0148z_1 + 0.3565
```



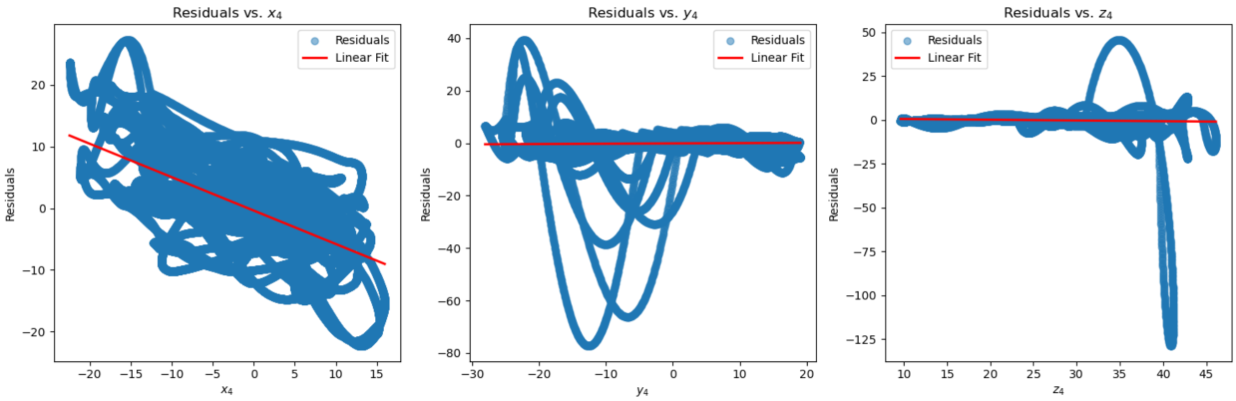
```
Equation for x_2: res_x_2 = 0.0413x_2 - 0.2386
Equation for y_2: res_y_2 = 0.0063y_2 + 0.0868
Equation for z_2: res_z_2 = 0.0102z_2 - 0.1991
```



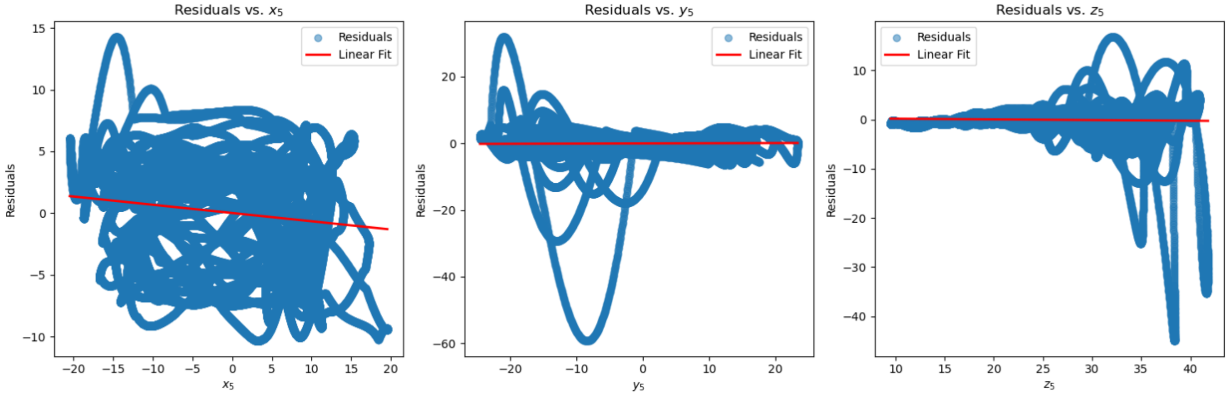

```
Equation for x_3: res_x_3 = 0.0258x_3 - 0.2359
Equation for y_3: res_y_3 = -0.0119y_3 - 0.0375
Equation for z_3: res_z_3 = 0.0037z_3 - 0.1280
```




```
Equation for x_4: res_x_4 = -0.5424x_4 - 0.3680
Equation for y_4: res_y_4 = 0.0118y_4 - 0.0970
Equation for z_4: res_z_4 = -0.0419z_4 + 0.9137
```



```
Equation for x_5: res_x_5 = -0.0668x_5 + 0.0081
Equation for y_5: res_y_5 = 0.0056y_5 - 0.0496
Equation for z_5: res_z_5 = -0.0138z_5 + 0.3258
```



From these graphs and fitted equations, we can infer that nodes 1 and 4 are hubs, while 2, 3 and 5 are low-degree nodes. Indeed, consider the following equation which we derived earlier:

$$\frac{\mathbf{x}_h(t+\Delta t) - \mathbf{x}_h(t)}{\Delta t} - \mathbf{f}(\mathbf{x}_h) \approx k_h \mathbf{V}(\mathbf{x}_h) + \mathbf{C}$$

We recall from lecture 20 that low-degree nodes in weakly coupled systems are assumed to follow the isolated dynamics with only small fluctuations due to weak interactions. Hubs, however, experience a larger cumulative effect from their connections. For nodes 1 and 4,

$$\frac{\mathbf{x}_1(t+\Delta t) - \mathbf{x}_1(t)}{\Delta t} - \mathbf{f}(\mathbf{x}_1) \approx k_1 \mathbf{V}(\mathbf{x}_1) + \mathbf{C} = (-x_1, 0, 0)$$

$$\frac{\mathbf{x}_4(t+\Delta t) - \mathbf{x}_4(t)}{\Delta t} - \mathbf{f}(\mathbf{x}_4) \approx k_4 \mathbf{V}(\mathbf{x}_4) + \mathbf{C} = (-0.5x_4, 0, 0)$$

and by equating coefficients, we infer that $C = (0, 0, 0)$, but more importantly, the system is x-coupled because only the x_1 and x_4 coefficients in the above graphs are significant (the other nonzero coefficients are negligibly small), which means that

$$V(\mathbf{x}_h) = V(x_h, y_h, z_h) = (\tilde{V}(x_h), 0, 0)$$

for some function $\tilde{V}(x_h)$ that depends linearly on x_h.

We will now estimate the coupling function $H(\mathbf{x}_i, \mathbf{x}_j)$ and the weights w_{ij}. As shown in the lecture,

$$\sum_{j=1}^{n} w_{ij} \mathbf{H}(\mathbf{x}_i, \mathbf{x}_j) \approx k_i \mathbf{V}(\mathbf{x}_i) + \mathbf{C} = k_i \mathbf{V}(\mathbf{x}_i)$$

and since we know that $V(\mathbf{x}_h)$ depends linearly on x_h through $\tilde{V}(x_h)$, and we are told that the coupling function is diffusive, it makes sense to test if $H(\mathbf{x}_i, \mathbf{x}_j) = \alpha(x_j - x_i, 0, 0)$ where α is the coupling strength. I estimated the unnormalised weights for node h, $\tilde{\mathbf{w}}_h = (\tilde{w}_{h1}, ..., \tilde{w}_{h5})$ where $\tilde{w}_{ij} = \alpha w_{ij} = \alpha A_{ij}$, where A_{ij} denotes the (ij)th entry of the adjacency matrix A. α is the coupling strength, which we are told is identical for all connections. By solving the linear system below via SINDy (again, because the coupling strength is the same for all connections, so any small weights should be set to zero):

$$\begin{bmatrix} H(\mathbf{x}_h(t_1), \mathbf{x}_1(t_1)) & \dots & H(\mathbf{x}_h(t_1), \mathbf{x}_5(t_1)) \\ \vdots & \ddots & \vdots \\ H(\mathbf{x}_h(t_T), \mathbf{x}_1(t_T)) & \dots & H(\mathbf{x}_h(t_T), \mathbf{x}_5(t_T)) \end{bmatrix} \begin{bmatrix} w_{h1} \\ \vdots \\ w_{h5} \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_{3T} \end{bmatrix}$$

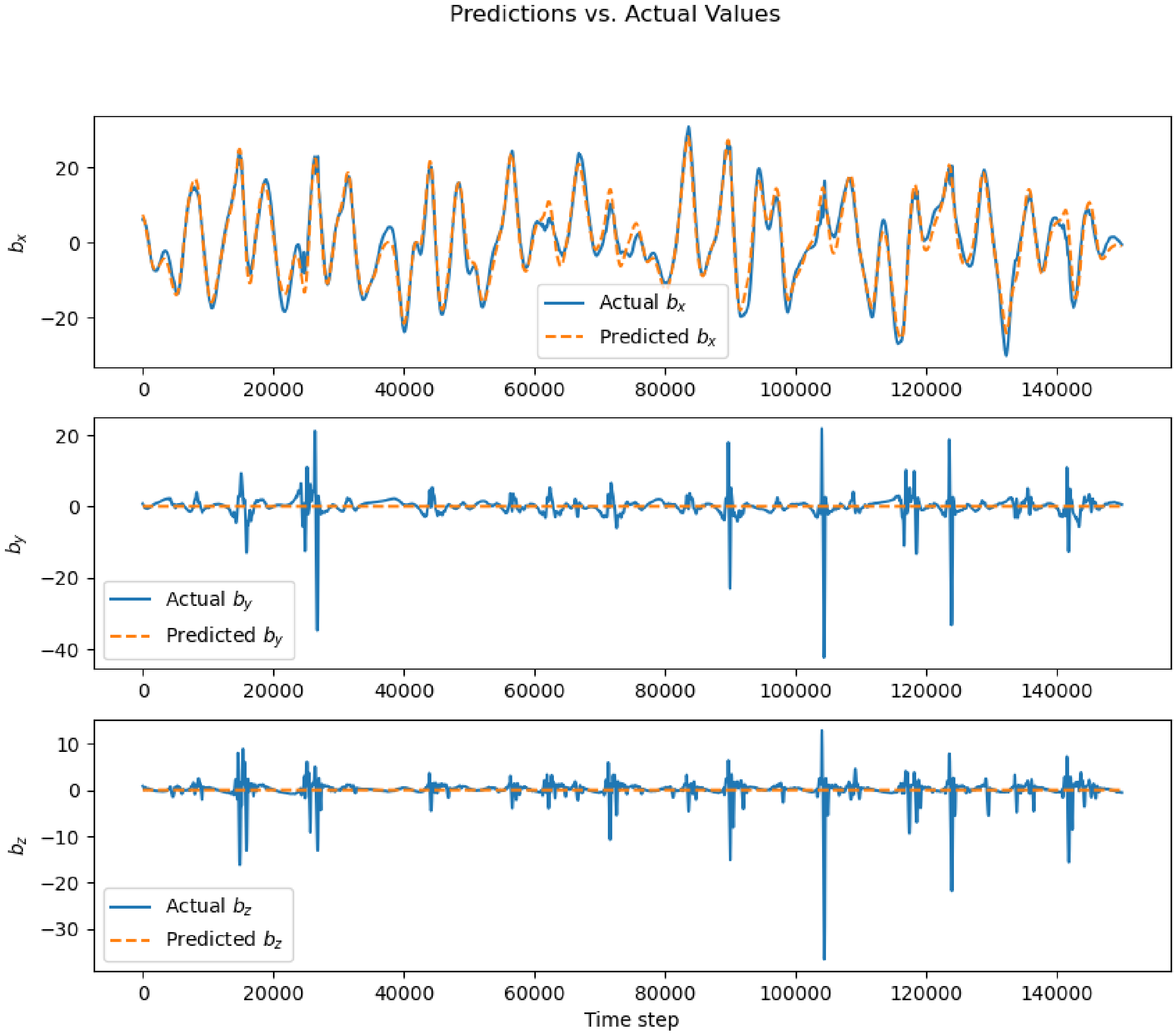
where $(b_{3k}, b_{3k+1}, b_{3k+2}) = \frac{\mathbf{x}_h(t+\Delta t) - \mathbf{x}_h(t)}{\Delta t} - \mathbf{f}(\mathbf{x}_h)$ for $k = 1, 2, ..., T$. My code for this is shown below.

```python
import numpy as np
import matplotlib.pyplot as plt

T = 150000  # Number of time steps
N = 5  # Number of weights

# Data
X = np.stack([time_series[i][:T, :] for i in range(1, 6)], axis=0)  # Shape: (5, T, 3)


# Construct H matrix
H = np.zeros((T-1, 3, N))

for t in range(T-1):
    for j in range(N):
        H[t, 0, j] = X[j, t, 0] - X[h-1, t, 0]  # Affect x component


# Reshape H to (3(T-1), N) for least squares
H_reshaped = H.reshape(3*(T-1), N)  # Now each row is a constraint

# residual vector
b = residual_matrix.reshape(3*(T-1), 1)


def sindy(A, b, threshold=0.15, iterations=15):
    """Performs sparse regression using iterative thresholding."""
    coeffs = np.zeros((A.shape[1], b.shape[1]))
    for i in tqdm(range(b.shape[1]), desc="Fitting SINDy models"):
        x = np.linalg.pinv(A) @ b[:, i]
        for _ in range(iterations):
            small = np.abs(x) < threshold
            x[small] = 0
            if np.any(~small):
                x[~small] = np.linalg.pinv(A[:, ~small]) @ b[:, i]
        coeffs[:, i] = x
    return coeffs



# Solve for w using SINDy
w = sindy(H_reshaped, b)

# Print result
print("Estimated weights:", w)

# Compute predictions: H_reshaped * w
predictions = H_reshaped @ w  # Shape: (T*3,)

# Reshape predictions and b back to (T, 3) for plotting
predictions = predictions.reshape((T-1, 3))


# Plot predictions vs. actual values for each component
```

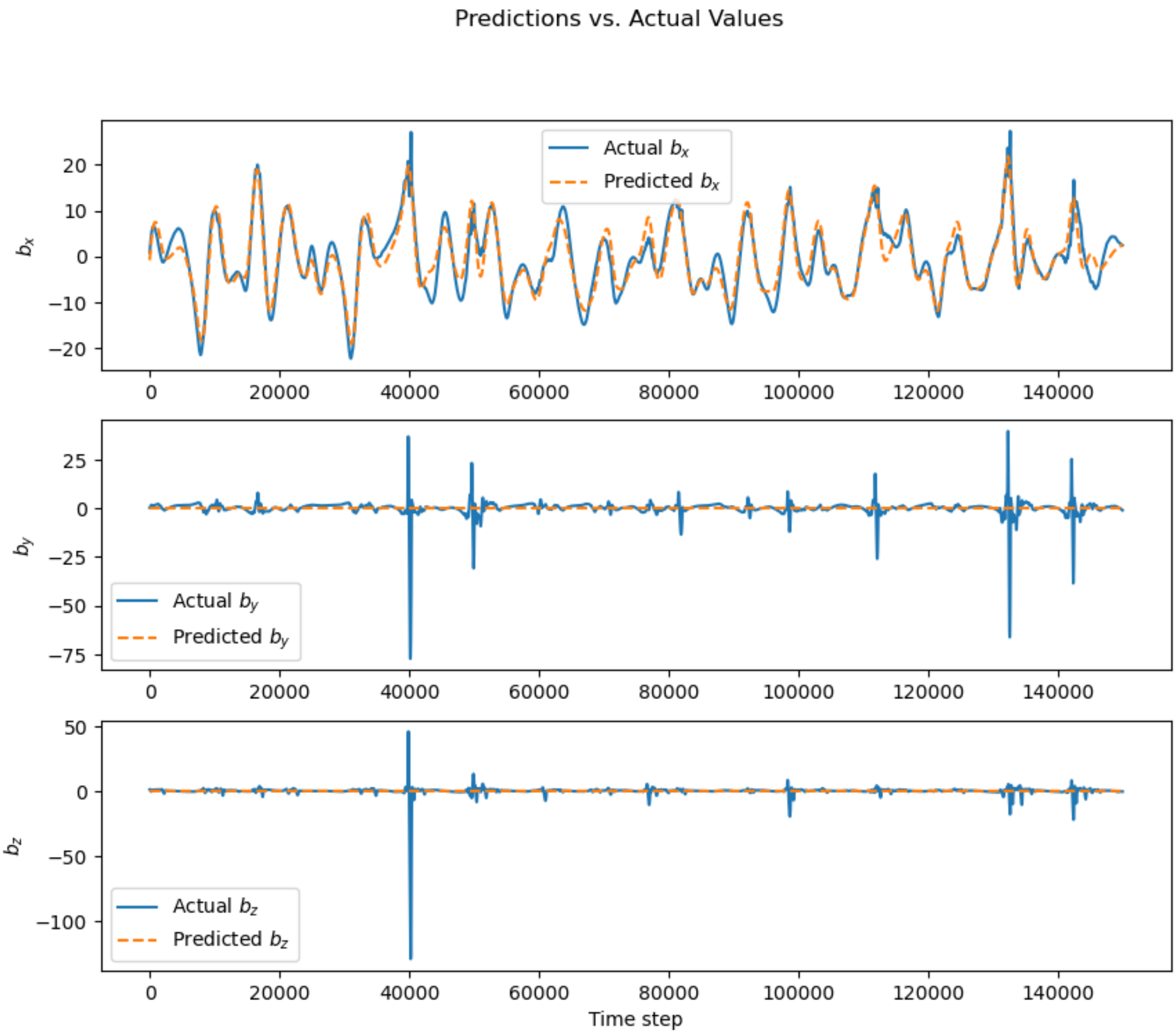
```python
fig, axes = plt.subplots(3, 1, figsize=(10, 8))

time_steps = np.arange(T-1)  # Correct time scale


for i, label in enumerate([r"$b_x$", r"$b_y$", r"$b_z$"]):
    axes[i].plot(time_steps, b[i::3], label=f"Actual {label}")
    axes[i].plot(time_steps, predictions[:, i], label=f"Predicted {label}",
    ↪  linestyle="dashed")
    axes[i].set_ylabel(label)  # Properly formatted subscript
    axes[i].legend()

plt.xlabel("Time step")
plt.suptitle("Predictions vs. Actual Values")
plt.show()

```

I obtained the following results for node 1 and 4 respectively:

```
Fitting SINDy models: 100%|████████████████████████████████████████| 1/1 [00:02<00:00,  2.67s/it]
Estimated weights: [[0.        ]
 [0.35225113]
 [0.36149755]
 [0.37849332]
 [0.        ]]
```



```
Fitting SINDy models: 100%|████████████████████████████████████████| 1/1 [00:00<00:00,  1.35it/s]
Estimated weights: [[0.34582815]
 [0.        ]
 [0.        ]
 [0.        ]
 [0.29885622]]
```



Looking at which weights are nonzero, we see that node 1 is coupled with nodes 2, 3 and 4, while node 4 is coupled with nodes 1 and 5. Since the coupling function is diffusive, by definition, $H(\mathbf{x}_i, \mathbf{x}_j) = \alpha(x_j - x_i, 0, 0)$ so if node i is coupled with node j, then node j is also coupled with node i. The adjacency matrix A is therefore

$$A = \begin{bmatrix} 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

I quantified the accuracy by computing the mean absolute errors (MAE) for each component. The code and MAE for hub node $h = 1$ are shown on the following page.

```python
from sklearn.metrics import mean_absolute_error

# Compute Mean Absolute Error (MAE) for each component
mae_x = mean_absolute_error(b[::3], predictions[:, 0])
mae_y = mean_absolute_error(b[1::3], predictions[:, 1])
mae_z = mean_absolute_error(b[2::3], predictions[:, 2])

print(f"MAE for x-component: {mae_x:.4f}")
print(f"MAE for y-component: {mae_y:.4f}")
print(f"MAE for z-component: {mae_z:.4f}")
```

MAE for x-component: 1.8332
MAE for y-component: 1.5327
MAE for z-component: 0.9685

Considering that the x-component generally takes values in the range [-20, 20], an absolute error of 1.83 is quite low. From the estimated weights for hub nodes 1 and 4, we see that the coupling strength is $\alpha = 1/3$. In this project, I have used ChatGPT to improve the structure of my code.